

De 0 a Python

By Kenkeiras





De 0 a Python

El objetivo de este cuaderno es presentar los conceptos básicos relacionados con la programación de software tales como búcles, variables, funciones e incluso hilos de ejecución , de forma compacta (para los que ven un manual de 300 páginas y les da un bajón =D), todo utilizando el lenguaje de programación Python como base.

1. Introducción

1.1 ¿Que es Python?

Python es un lenguaje de programación multiplataforma e interpretado, esto quiere decir:

- Que funciona en casi cualquier sistema operativo sin necesidad de modificar absolutamente nada.

- Que el código del programa se ejecuta directamente a traves de un programa "intérprete" (llamado así porque "interpreta" el código), en vez de pasar por una fase de compilación para convertirlo en un archivo ejecutable (como los famosos .exe), permitiendo incluso escribir el código a medida que se ejecuta, de forma interactiva.

Este lenguaje se programa con un "editor de texto plano", es decir, algo tan simple como el Bloc de notas o Notepad++ en Windows o gEdit o kWrite en GNU/Linux, el própio intérprete incluye un editor alternativo para esta tarea (sí, el WordPad o el MS Word no funcionan aquí).

Es importante saber que Notepad++, gEdit, kWrite y el editor que incluye el intérprete facilitan el trabajo coloreando el código para distinguir mejor las cosas.

2.Instalación, uso del intérprete y Hola mundo

Nota: por compatibilidad se utilizará la versión por defecto en debian al momento de escribir esto (2.6)

2.1 Instalación

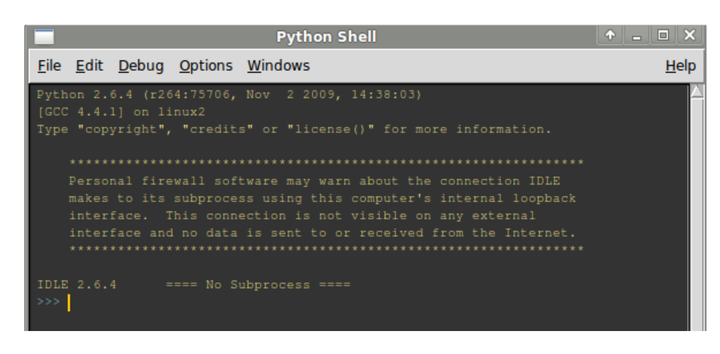
En Windows:
El archivo para instalarlo se puede encontrar en http://www.python.org/download/releases/2.6.6 más concretamente, http://www.python.org/ftp/python/2.6.6/python-2.6.6.msi (se asume que el lector es capaz de instalar un programa sencillo ;)

En otros sistemas (Gnu/Linux, *BSD, ...):
Lo mas probable es que ya este instalado, de lo contrario se puede hacer a través del sistema de paquetes de la distribucion o a través de las fuentes en http://www.python.org/ftp/python/2.6.6/Python-2.6.6.tgz
Con la instalación de Windows se incluye un interfaz gráfica para el intérprete, en otros sistemas hay que buscar el paquete "idle" (en Synaptic está en la sección "Lenguaje de

programación Python")

2.2 Uso del intérprete

Una vez abierto el intérprete aparecera algo así:



Esto nos permitirá usarlo de dos formas, introduciendo el código manualmente (como si fuera una consola de comandos) o ejecutar archivos que contengan el código.

Para la primera forma (introducir el código manualmente), solo hay que escribirlo despues del ">>>".
Si se utliliza la segunda forma, se puede abrir un archivo a través de "**Archivo**"(o "**File**") y "**Abrir**" (o "**Open**") para crear un nuevo archivo utilizaremos "**Nueva ventana**" (o "**New window**") en vez de "Abrir", con lo que obtendremos una ventana en la que podremos editar el archivo, y finalmente ejecutarlo con "F5".

Nota: si el código incluye caracteres que no existen en el estándar ASCII (como la "ñ") se producirá un error, esto se puede solucionar con este comentario al principio del código:

```
# -*- encoding: utf-8 -*-
```

Lo que hará que se utilice UTF-8, mucho más amplio.

2.3 Hola mundo

A modo de ejemplo y para comprobar que todo funciona bien, escribiremos en un archivo (o en el intérprete)

```
print 'Hola Mundo!'
```

y lo ejecutamos, esto mostrara el mensaje:

```
Hola mundo!
```

2.4 Comentarios

Es posible introducir comentarios en archivos de python, usando el carácter "#" al principio del comentario, de esta forma el intérprete simplemente se saltará el resto de la línea (si hay que comentar varias líneas, se repetira esto en cada línea), por ejemplo:

```
#Esto es un ejemplo de hola mundo
print 'Hola mundo!'
# Facil, verdad?
```

Seguirá mostrando:

```
Hola mundo!
```

3. Variables y entrada/salida de datos

Para utilizar una variable en python simplemente hay que asignarle un valor, por ejemplo:

```
nombre='Joe'
```

Esto creara una variable llamada nombre, a la que se le asignara el valor "Joe", las variables que contienen cadenas de caracteres (a partir de ahora "strings") deben estar entre "," o ',' , no pasa lo mismo con las variables que almacenan números, por ejemplo:

```
i=10
```

Ese comando creara una variable llamada i con el valor 10.







Nota: Hay que tener en cuenta que introducir espacios en cualquier parte del código (a menos que sea dentro de un string) no afecta a como se ejecutará, por ejemplo.

```
i=10
```

es igual a :

```
i = 10
```

Esto se puede usar para hacer el código menos compacto y más legible.

Se pueden reutilizar nombres de variables, por ejemplo

```
i=200
```

Haría que el valor de i fuese 200, y

```
i='xyz'
```

Cambiaria el valor i por el string “xyz”

Además se pueden hacer variables con el valor de otras, por ejemplo si hacemos

```
var=i
```

A la variable var se le asigna el valor de la variable i.

Se puede crear una variable con cualquier secuencia de letras , numeros y ciertos caracteres (como _), siempre que ni empiece por un numero ni que coincida con el nombre de una palabra clave, como un comando o una expresión del lenguaje (lógicamente no se puede crear una variable llamada = :P).

Nota: Es importante destacar que los comandos se acaban con un salto de línea, a diferencia de otros lenguajes de programacion, que acaban con un “;”, que no se usa en python.

Pero, si no se puede interactuar con el usuario no hay diversión, ¿verdad?

Para mostrar algo por pantalla usaremos el comando print y despues lo que queramos imprimir (variables, numeros o strings), separados por comas, por ejemplo los comandos

```
nombre='Joe'
print 'Hola',nombre
```

Mostrarán

```
Hola Joe
```

(Y el cursor quedaría aquí)

Como se puede ver, el comando print imprime lo que hayamos introducido en una línea y después pasa a la siguiente, esto a veces no es lo que queremos, en esos momentos acabaremos el comando con una coma (sin nada después).

Para permitir al usuario introducir datos existen dos comandos: input y raw_input

Para utilizarlos se usa input(string) siendo el string lo que quieras que se muestre al usuario justo antes de que introduzca el dato, por ejemplo:

```
nombre=raw_input('Hola, como te
llamas? ')
```

Esto le dirá al usuario “Hola, como te llamas?” y lo que escriba el usuario lo almacenará en la variable nombre.

La diferencia entre input y raw_input es que el primero pasa el valor directamente (así que si en el ejemplo anterior usamos input en vez de raw_input el usuario tendria que introducir el nombre entre comillas), y raw_input convierte todo en un string antes de meterlo en la

variable, esto se tratará mas afondo en el siguiente capítulo.

Ejemplo:

```
nombre=raw_input('Hola, como te
llamas?')
print 'Hola',nombre
```

Las strings aceptan cadenas de caracteres especiales que permiten darle formato al texto si se definen entre "," (pero no entre ',') , van después de un \

Caracteres	Descripción
\n	Salto de línea (pasa a la siguiente línea)
\t	Tabulador
\\	\
\"	" (sin acabar el string, si fuese el caso)
\'	' (sin acabar el string, si fuese el caso)

Sin embargo, si el programa se usa desde la consola, y no desde el intérprete con interfaz gráfica se permiten usar algunos más:

Caracteres	Descripcion
\r	Retorno de carro (el cursor vuelve al principio de la línea)
\b	Retorno (el cursor retrocede una posición)
\v	Tabulador vertical

4. Tipos de variables

En python existen varios tipos de variables, aqui se mostraran los siguientes

- Números
- Strings (cadenas de caracteres)
- Caracteres individuales
- Booleanos
- Listas
- Diccionarios

Además existe una variable "especial" llamada None ("ninguna") con su propio tipo (NoneType).

Para averiguar el tipo de una variable se usa el comando

```
type(variable)
```

4.1 Variables numericas

Las variables numericas se pueden declarar de varias formas, como enteros normales(**int**), hexadecimales (**hex**) binarios (**bin**) o con decimales, punto flotante(**float**), para pasar de uno a otro solo hay que hacer lo siguiente. En realidad los binarios y hexadecimales se convierten automáticamente a enteros (la conversión de enteros a hexadecimales o binarios en realidad genera strings).

Un ejemplo de la conversión entre varios tipos de enteros:

```
#  Creamos una variable numerica
#  decimal
num1=10

#  Creamos una variable numerica
#  hexadecimal (hay que añadir "0x"
#  antes del numero)
num2=0x10


#  Creamos otra variable con el
#  valor de num1 pasado a
#  hexadecimal,usando hex()
num3=hex(num1)

#  Creamos la ultima variable con
#  el valor de num2 pasado a
#  entero, usando int(), o float()
#  si contiene decimales
num4=int(num2)
```

```
#Mostramos las variables
print 'Num1:',num1
print 'Num2:',num2
print 'Num3:',num3
print 'Num4:',num4
```

Esto mostrará lo siguiente:

```
Num1: 10
Num2: 16
Num3: 0xa
Num4: 16
```

Esto muestra que el valor de una variable hexadecimal es convertida automaticamente a entero después de ser creada, por eso la variable num2 (el valor original) y num4 (el valor convertido a entero) son iguales, y por eso la variable num3 se muestra como un string (para no ser convertida a decimal de nuevo).

Los números se operan como se harían naturalmente, enteros y flotantes pueden operar entre ellos sin ningún problema, pero dado que operar dos enteros da como resultado un entero, (si al menos uno de los dos es flotante el resultado también lo será), esto se nota especialmente en las divisiones:

```
>>> 10/3
3
>>> 10/3. # Un 3 de punto flotante
3.3333333333333335
>>> 1+1
2
>>> 1+1.
2.0
>>> 2/2
1
>>> 1/2
0
>>> 1/2.
0.5
>>> 2*5
10
>>> 1-2
-1
```

4.2 Strings

Los strings son cadenas de caracteres, para pasar una variable cualquiera a string se utiliza

```
str(variable)
```

Esto es útil para crear cadenas de texto, para concatenar varias cadenas de texto se hace:

```
# Creamos una cadena
cadena1='abcdef'

# Creamos otra cadena
cadena2='ghijklm'

# Juntamos las dos en otra
cadena3=cadena1 + cadena2

print cadena3
```

Mostrará

```
abcdefghijklm
```

Para saber si el string contiene a otro se hace así:

```
cadena1='qwertyuiop'
cadena2='er'
contenida=cadena2 in cadena1
print contenida
```

Mostrará

```
True
```

Es decir, "verdad", en caso de que la segunda cadena no este en la primera mostraría False

Otra cosa importante sobre los strings es conocer su longitud, esto se hace con len(cadena)

```
cadena='123456789'
longitud=len(cadena)
```

```
print longitud
```

También se puede repetir una cadena varias veces haciendo cadena * veces ,por ejemplo:

```
cadena = 'bla'
print cadena * 3
```

Mostrará

```
blablabla
```

Además se puede extraer una subcadena con cadena[inicio : fin] (hay que considerar que la primera posición es la 0) por ejemplo:

```
cadena = 'abcde'
subcadena = cadena [1:3]
print subcadena
```

Mostrará

```
bc
```

4.3 Caracteres

Los caracteres strings de longitud 1, para extraer uno se utiliza cadena[posición] :

```
cadena='abcdefg'
caracter=cadena[0]
print caracter ,
caracter=cadena[1]
print caracter ,
caracter=cadena[-1]
print caracter ,
```

Esto mostraría:

```
a b g
```

Por que las cadenas se leen desde la posicion numero 0, que es la primera, y se leen al revés (del final al principio) si se hace con números negativos

Para obtener el código asignado a un caracter se utiliza ord(caracter), por ejemplo

```
cadena=raw_input('Introduce una
letra: ')
caracter=cadena[0]
numero=ord(caracter)
print 'El caracter',caracter,'es el
numero',numero
```

Para hacer la operacion al revés (obtener un caracter a partir de su numero), se utiliza chr(número), por ejemplo:

```
cadena=raw_input('Introduce un
numero: ')
numero=int(cadena)
caracter=chr(numero)
print 'El
numero',numero,'corresponde al
caracter',caracter
```

4.4 Booleanos

Este tipo de variables solo tiene dos valores posibles: Verdadero (True) o falso (False) y se utilizan ampliamente con los condicionales (asi que se explicara su uso con ellos, más tarde).

4.5 Listas

Las listas son (como su nombre indica) listas donde se pueden almacenar valores de cualquier tipo (incluso otras listas), se accede a los valores que hay dentro de ellas de la misma forma que los caracteres lo hacen con los strings lista[inicio : fin] , lista[posicion], de hecho la función len() tiene el mismo efecto en los dos.

Nota: La primera posición sigue siendo la número 0, no la 1, como se podría esperar

Para crear un nuevo elemento al final de la lista se hace lista1.append(valor), para quitar un valor se hace lista1.pop(posicion) (lista1 sería el

nombre de la variable lista)

Una forma sencilla de construir listas de números es con range(num1,num2), que construiría una con todos los valores de num1 hasta num2 (excluyendo al último).

Por ejemplo:

```
lista = range(0, 10)
print lista
```

Mostrará:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4.6 Diccionarios

Los diccionarios son estructuras que relacionan un objeto (string, número ...) con otro cualquiera, para visualizarlo mejor:

```
# Crea un diccionario vacío
d_ejemplo = {}

# Añade una entrada para 1
d_ejemplo[1] = 'Blah!'
# Añade una entrada para 'nombre'
d_ejemplo['nombre'] = 'Joe'
# Añade otra entrada más
d_ejemplo['numero'] = 42

# Ahora muestra lo que hay

# Muestra 'Blah!'
print d_ejemplo[1]
# Muestra 'Joe'
print d_ejemplo['nombre']
# Muestra 42
print d_ejemplo['numero']

# Se crea un diccionario con dos
# entradas
d_ej2 = {1: 'Blah**2!', 'numero':
32}
print d_ej2[1] # Muestra 'Blah**2!'
```

```
# Ahora veamos que pasa si no existe
print d_ej2['no_existe']
# Produce un error
```

5. Condicionales, bucles y funciones

Para modificar el orden de ejecucion de un programa se utilizan condicionales (que permiten hacer cosas distintas segun se dé una condición o no) y los bucles (en los que se repite varias veces una parte del código)

En python el código que pertenece a un condicional, bucle, etc... debe seguir esta estructura

```
condicional
        código
        código
        condicional
                código
                ...
                código
        código
        ...
```

Es decir, que los espacios antes de algún comando sirven para determinar a donde corresponde (las líneas en blanco se usan para hacer el código mas legible)

5.1 Condicionales

Para decidir si una parte del programa se ejecuta o no se utiliza las funciones if (si), elif (o-si) y else (sino)
Ejemplo:

```
numero=raw_input('Cuanto es 1+1? ')

# Posibilidad 1: El usuario ha
# pulsado directamente Enter
if (len(numero)==0):
        print 'No has introducido nada'
```

```
# Posibilidad 2: El usuario se ha
# equivocado
elif(numero!='2'):
      print 'No, te has equivocado'

# Posibilidad 3: El usuario ha
# acertado
else:
      print 'Si, has acertado'
```

Este código hace lo siguiente: pregunta al usuario cuanto es 1+1, si el usuario no dice nada, el programa sigue por print "No has introducido nada", si no se da la condición, se prueba con lo sigiente, si la condición se cumple y el usuario se ha equivocado, el programa ejecuta print "No, te has equivocado", sino solo queda una posibilidad y el programa muestra "Si, has acertado"

Nota: cualquier operador en python se puede agrupar en paréntesis para cambiar el orden de prioridades, de la misma forma que se hace con las matemáticas convencionales.

El funcionamiento es asi:

```
if (condición):
      ...
elif (condición):
      ...
elif (condición):
      ...
              ...
else:
      ...
```

El if (si en inglés) ejecuta el código que contiene si la condición resulta verdadera, sino, si a continuación hay algun elif (o-si en inglés) se comprueba si se da condición (si es asi se ejecuta el código), sino se vuelve a comprobar si hay otro elif, así hasta que se encuentre alguno verdadero o hasta que no quede ninguno, después, si no se ha cumplido ninguno, se ve si hay algún else (sino en inglés), y si lo hay se ejecuta su código.

Para comprobar las condiciones existen las siguientes operaciones:

Condición	Código
Dos variables iguales	`variable1 == variable2` (no confundir con = !!)
Dos variables distintas	`variable1 != variable2`
Una variable mayor que otra	`variable1 > variable2`
Una variable mayor o igual que otra	`variable1 >= variable2`
Una variable menor que otra	`variable1 < variable2`
Una variable menor o igual que otra	`variable1 <= variable2`
Dos condiciones a la vez	`(condición1) and (condición2)`
Al menos una condición	`(condición1) or (condición2)`
El contrario de una condición	`not condición`

Las condiciones son en realidad variables de tipo booleano, asi que las operaciones se aplican a los booleanos también.

5.2 Bucles

Hay dos tipos de bucles: bucles while y bucles for

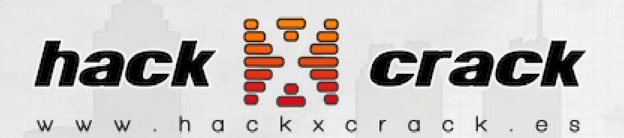
Los bucles while (mientras en inglés) ejecutan un código mientras se dé la condición, por ejemplo:

```
i=10
while i>=0:
      print  i
      i=i-1
print 'Despegue!'
```

Mostrará números mientras i sea mayor o igual a 0

```
10
9
8
7
6
5
4
```








```
3
2
1
0
Despegue!
```

Así que para hacer que algo se ejecute continuamente hariamos algo así:

```
while True:
      print 'Falta mucho?'
```

Que mostrará continuamente:

```
Falta mucho?
Falta mucho?
Falta mucho?
Falta mucho?
Falta mucho?
```

...

Esto pasa por que la condición es siempre True (o Verdad)

Los bucles for, ejecutan el código por cada elemento en una lista (o string), por ejemplo:

```
for elemento in lista:
      print elemento
```

Que ejecutará el código con cada posicion de la lista (o del string), poniendo lo que haya en esa posicion en la variable, por ejemplo:

```
# Mostrar todas las letras de una
# palabra
palabra=raw_input('Introduce una
palabra: ')
for letra in palabra:
      print 'Letra',letra
```

Mostrará una letra de la palabra que se introduzca en cada línea, de la primera a la última.

```
# Mostrar todos los numeros del 10
# al 0
for numero in
[10,9,8,7,6,5,4,3,2,1,0]:
      print numero
```

Mostrará:

```
10
9
8
7
6
5
4
3
2
1
0
```

Como se puede ver sigue el orden presente en la lista o en la cadena.

Hay dos expresiones que permiten modificar el comportamiento dentro de los bucles (siempre con respecto al mas interno), break, que sale del bucle y continue que salta al final.

Por ejemplo:

```
i = 0
while i < 10:
      print i
      i = i + 1
```

Podría ser substituido por

```
i = 0
while True:
     if (i >= 10):
          break
     print i
     i = i + 1
```

5.3 Funciones

Se puede extender el lenguaje Python definiendo funciones propias funciones propias, esto se hace así:

```
def nombre_de_la_función
(parametro1,parametro2,...):
      código
```

Los parámetros son variables que puede utilizar la funcion, se puede hacer que algunos sean opcionales dandoles una valor por defecto, por ejemplo:

```
def función_de_ejemplo
(parámetro1,parámetro2=24):
      código
      código
      ...
```

La estructura del código de una función es igual a la de los condicionales:

```
def función():
      código
      código
      código
      ...
      condicional:
            código_del_condicional
            código_del_condicional
            código_del_condicional
            código_del_condicional
      código
      código
```

Las funciones pueden devolver algun valor, usando return , seguido por el valor a devolver , por ejemplo la siguiente función hace la suma de dos numeros:

```
def suma (num1,num2):
      resultado=num1+num2
      return resultado
```

```
# Un ejemplo de su uso sería
numero1=2
numero2=7
numero3=suma(numero1,numero2)
print 'La suma es',numero3
```

Mostraría la suma de 2 y 7:

```
La suma es 9
```

6. Uso de librerías

Las librerias son archivos que contienen conjuntos de funciones, por ejemplo hay algunas que permiten hacer interfaces graficas, otras que permiten manejar conexiones de red y otras que permiten usar funciones matematicas más complejas.

Para usar una librería hay que "importarla", esto se hace de esta forma:

```
import math
```

Esto importa la libreria "math", que contiene funciones matematicas, pero para llamar a una función hay que hacerlo de esta forma:

```
libreria.función()
```

Esto puede resultar incómodo si la función se utiliza muchas veces, para esto se utiliza el siguiente código:

```
from libreria import función
```

Asi se puede llamar a función directamente, otra opción es importar de una vez todas las funciones de la librería:

```
from libreria import *
```

Un@ puede hacer su propia librería guardando el código (funciones y demás) en un archivo y dejándolo en el directorio en el que está el que







lo importa, si el archivo es lib1.py, se importaria con import lib1

Aprovecho para comentar algunas funciones y librerías básicas:

6.1 Librería os

Para lanzar un comando de systema (uno de la shell) se hace de esta forma:

```
import os
os.system('comando')
```

Por ejemplo:

```
import os
os.system('echo Hola')
```

Lanzara el comando de shell “echo Hola”

Para saber el directorio actual se utiliza os.getcwd(),por ejemplo:

```
import os
print os.getcwd()
```

Y para cambiar de directorio os.chdir(“directorio”)

```
import os
os.chdir('directorio')
```

6.2 La libreria sys

La característica mas destacable de la librería sys es la posibilidad de manejar los parámetros del programa en el comando que los lanzo, por ejemplo:

```
import sys
print sys.argv[0]
```

Mostrará el nombre del archivo (el parámetro 0).

Esto es útil cuando se quieren hacer programas que puedan ser lanzados desde scripts de consola, para evitar que el usuario tenga que introducir los datos a mano, por ejemplo:

```
import sys
argc=len(sys.argv)
print 'Hay', argc, 'argumentos'
for i in sys.argv:
      print i
```

Esto mostraría el número de argumentos y estos. Además da el control de 3 archivos, el de entrada (sys.stdin), que muestra lo que se introduce por el teclado, el de salida (sys.stdout) donde se escribe lo que saldrá por pantalla y el de errores (sys.stderr), ahora mismo se hablará de como manejar los archivos.

7. Manejo de archivos

Empezamos con open, esta función sirve para abrir un archivo, se le pasan dos parametros, el nombre del archivo y las opciones (dependiendo de si solo puede leer ,si puede escribir,...)

Opción	Código de opcion
Leer	*r*
Añadir a continuacion	*a*
Escribir	*w*

La opcion de lectura nunca crea un archivo nuevo.
La opcion de añadir a continuacion crea un archivo nuevo si no hay ninguno.
La opcion de escribir crea un archivo nuevo (y borra antes el otro si habia alguno).

Opción	Código de opcion
Archivo binario (puede contener caracteres no legibles)	*b*
Archivo de texto (se adaptarán los saltos de línea a los del SO)	*t*

Estas dos se combinan para elegir distintas

opciones, por ejemplo “rb” significa que es de lectura y es binario.
Si después de las opciones hay un +, el archivo se puede manejar de todas las formas (leer y escribir)
Ejemplo:

```
archivo=open('archivo1','at+')
```

Abre un archivo llamado “archivo1” con permisos para todo y si no existe lo crea.

Para leer algo de un archivo se hace archivo.read(numero_de_carateres_a_leer) si no se especifica la cantidad de caracteres a leer, se lee todo el archivo, una alternativa es archivo.readline(),que lee una línea del archivo.

Nota: si archivo.read() o archivo.readline() devuelve un string de longitud 0, es que se acabó el archivo.

Para escribir algo en un archivo se hace archivo.write(“string de ejemplo”), que escribe string de ejemplo en el archivo.

Por ultimo, para cerrar un archivo se utiliza archivo.close().

Adicionalmente existe archivo.flush() que hace que los cambios en el archivo se guarden, esto es especialmente útil si se manejan los archivos sys.stdout y sys.stderr a mano, ya que de otra forma no muestran los cambios hasta que se acabe la línea.
Ejemplo (Un programa que copia un archivo):

```
# Debe ser binario por si acaso
inp=open('archivo1','rb')
# Lo mismo con este
outp=open('archivo2','wb')
string=inp.read(1)
while len(string)>0:
        outp.write(string)
        string=inp.read(1)
inp.close()
outp.close()
print 'Archivo copiado! :)'
```

8. Manejo de errores

A veces hay partes del código que puden generar errores, al importar librerias que no son estándar, o al intentar leer un archivo que no existe, en esos momentos habrá que tener algunas precauciones para evitar que el código falle sin recuperacion posible, para eso se utilizan las funciones try, exception y raise

Las funciones try y exception funcionan de una forma parecida a if y else:

```
try:
        f=open('archivo','r')

except:
        print 'No se ha podido abrir
el archivo'
```

Esto hará que python intente abrir el archivo y en caso de error imprima “No se ha podido abrir el archivo”.
Pueden utilizarse varias líneas dentro del try o del exception, para hacer el código mas corto por ejemplo:

```
try:
        import libreria_no_estándar
        a=input('Introduce un numero
del 1 al 10')

except:
        print 'Se ha producido un
error'
```

Aqui hay dos líneas que puden causar el error, importar una librería que no siempre esta disponible o usar input, que produce un error si el usuario introduce caracteres alfabeticos, en los dos casos se mostrara por pantalla <<Se ha producido un error>>

Por ultimo esta raise(tipo de error,"Descripcion del error") que sirve para avisar de un error, el tipo de error mas frecuente es el Exception,por ejemplo supongamos que queremos hacer una función de suma que no acepte numeros negativos:

```
def suma(a,b):
      if (a<0) or (b<0):
            raise(Exception,'No se
permiten numeros negativos')
      else:
            return a+b
```

En algunos casos conviene conocer una operación relativamente poco común, pass que no hace nada, su objetivo es evitar errores cuando no hay nada que se deba poner en el except.

```
try:
    import psycho.full
except:
    pass
```

En este caso se intenta importar la librería psycho.full (que hace que el código se ejecute más rápidamente por explicarlo de forma rápida) pero no hace nada sino falla, ya que no hace falta.

9. Conexiones de red

Ahora veremos como realizar conexiones a través de internet y como enviar y recibir datos a través de ellas, concretamente a través de conexiones TCP por ser los mas comunes y sencillos de utilizar.
El primer paso es importar la librería socket y crear el objeto que nos permitirá comunicarnos.

```
import socket
sock = socket.socket()
```

A partir de aquí hay dos opciones, conectarnos a una dirección identificada con una IP o un dominio y un puerto, o esperar a que alguien se conecte a nosotros, primero veremos como conectarnos, para hacer esto simplemente hay que invocar la función connect del socket que hemos creado.

```
# Nos conectaremos al puerto 80 de
# www.google.es
sock.connect(('www.google.es', 80))
# Hay que prestar atención a los
# paréntesis, ¡hay dos pares!
```

Y ya tenemos una conexión con el servidor web de www.google.es. Si queremos enviar una cadena hay que usar la función send del objeto que se conecto, por ejemplo, vamos a perdirle al servidor que nos envíe la página web.

```
# Le pedimos la página "/" (la que
# se usa por defecto)
sock.send("GET / HTTP/1.1\r\n")
# Y le decimos es de www.google.es,
# para evitar errores
sock.send("HOST:
www.google.es\r\n\r\n")
```

Ahora necesitamos leer lo que el servidor nos envió, para esto se usa la funcion recv indicandole cuantos caracteres queremos leer.

```
data  = sock.recv(1024)
# Le pedimos 1024 caracteres
print data
```

Cuando ya no necesitemos la conexión la cerramos con la función close, como se hace con los archivos.

```
sock.close()
```

Para recibir conexiones primero tenemos que asociar el socket a un puerto, con bind , con los parámetros direccion donde se aceptará ('' para cualquiera) y puerto de la misma forma que se

hace con connect.

```
escucha = socket.socket()
escucha.bind(('',1234))
  # Escuchamos en el puerto 1234
  # en cualquier interfaz
```

Ahora hay que decidir cuantas conexiones pueden estar a la espera de que sean aceptadas por nuestro script, esto se hace con listen , indicandole el número de conexiones, por ejemplo 10 (si hay un firewall activo, probablemente este sea el momento donde se produzca el aviso).

```
escucha.listen(10)
```

Por último solo queda aceptar las conexiones, esto se hace con accept, devuelve 2 parámetros, un nuevo socket para la conexión y la dirección desde donde viene la conexión, el socket que se utilizó para aceptar se puede (de hecho se debería) reutilizar para aceptar conexiones al mismo puerto, volviendo a llamar a accept. El socket que se obtiene de accept ya está conectado, se puede enviar y recibir datos directamente, por ejemplo, si lo que queremos es que envíe “Hola mundo!” a quién se conecte, y después cierre la conexión.

```
while True:
    # Se aceptan conexiones
    conn, addr = escucha.accept()
    # Se envía el mensaje
    conn.send('Hola mundo!')
    # Y se cierra la conexión
    conn.close()
```

10. Programación orientada a objetos

Python nos permite crear nuevos tipos de variables, a las que llama clases (class) permitiendo así practicar lo que se llama la programación orientada a objetos (OOP).

Las clases se declaran de forma similar a las funciones, la “palabra clave” es class, que va seguida de el nombre de la clase y ':'

```
class contador : # Clase de ejemplo
```

Una vez hecho esto, podemos incluir las variables y funciones que contendrá la clase, supongamos que queremos hacer un contador que nos muestre las veces que se llamo a una función, necesitamos una variable para llevar las cuentas

```
    veces = 0
```

Y una función a la que llamar, que incrementará el contador y mostrará las veces, para acceder a las variables comunes a todo el objeto se precede con self. , entonces la función quedaría así.

```
    # Siempre se pasará como primer
    # parámetro self
    def  cuenta(self):
                # Incrementa el
contador
        self.veces = self.veces + 1
        print self.veces
```

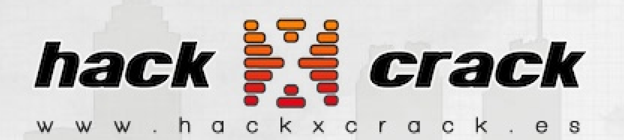
En nuestro ejemplo, la clase contador ya funcionaría

```
c = contador()
c.cuenta()
c.cuenta()
```

Mostrará

```
1
2
```







Eso todo lo necesario para clases básicas, pero hay unas cuantas funciones que permiten hacer

Función	Objetivo
`__init__(self, parámetro1, parámetro2 ...)`	Se llamará al crear el objeto, recibe los parámetros que se dan al crearlo
`__str__(self)`	Pide la conversión a string de la clase (suele usarse cuando se usa con **print**)
`__int__ (self)`	Pide la cinversión a número de la clase
`__float__ (self)`	Pide la conversión a punto flotante de la clase
`__cmp__ (self, otro)`	Compara el valor del objeto con **otro** , se devuelve menos que 0 para menor que el, 0 para igual y mayor de 0 para **self** > **otro**

a las clases hacer más cosas:

Además hay otra posibilidad única de la programación orientada a objetos, la herencia, que permite crear una nueva clase que parte de una anterior, esto se hace añadiendo la clase "base" entre paréntesis en la declaración de la nueva clase, después del nombre de esta y antes de los ':', veamos por ejemplo como extender el contador para que soporte las funciones como __init__ , __str__ , ...

```
class nuevo_contador (contador):
    def __init__(self, base = 0):
    # inicializa el contador
        self.veces = int(base)

    def __str__(self):
    # Devuelve la representación en
    # cadena
        return str(self.veces)

    def __int__(self):
    # Devuelve la representación en
    # número entero
        return self.veces

    def __float__(self):
    # Devuelve la representación en
    # número de punto flotante
        return float(self.veces)

    def __cmp__(self, otro):
    # Compara con otro valor
    # Negativo si el otro es mayor,
    # 0 si son iguales.
    # Positivo si el otro es menor.
        return self.veces - otro
```

Nota: No hay ningún problema en reescribir métodos que existieran en la clase inicial, a esto se le llama sobrecarga de método.

```
>>> c = nuevo_contador(10)
>>> c.cuenta()
11
>>> print c
11
>>> c == 11
True
>>> c.cuenta()
12
>>> c == 11
False
```

11. Programación multihilo básica

La programación multihilo permite que un script ejecute varias cosas a la vez, especialmente útil cuando hay cuellos de botella importantes, como las conexiones de red, si por ejemplo queremos hacer un escáner de puertos es mucho más eficiente lanzar unos cuantos hilos para realizar varias conexiones a la vez, con lo que el tiempo se reduce bastante. Sobre la coordinación de código en paralelo (categoría donde se engloba la multihilo) se dice

-¿Por que cruzó la gallina paralela la calle?
-cruzar la calle Para

-¿Por que cruzó la gallina paralela la calle?
-Para calle la cruzar

¿Que se quiere decir con esto?, que la programación en paralelo implica que mucho







código se va a invertir solamente en coordinar el programa... y bueno, a quien le guste esa parte, pues bien, pero a quien no...

La programación en paralelo en un mundo en sí mismo, así que apenas rasgaré la superficie explicando como crear un hilo de ejecución independiente en Python. Por otro lado no puedo dejar sin comentar que cada hilo es un objeto distinto ¡!, así que si no comprendiste bien el capítulo anterior, toca volver.

Para crear un hilo de ejecución necesitaremos una clase nueva, que se base en (herede) la clase que se puede encontrar en threading.Thread

```
from threading import Thread

class nuevo_hilo(Thread):
```

Si se sobreescribe (técnicamente, se sobrecarga) el __init__ de la clase, en la nueva función hay que llamar al __init__ original antes de nada, por ejemplo, si queremos que el hilo cuente hasta un número que digamos, y que se defina en el __init__ tendremos que hacer.

```
    def __init__(self, tope):
        Thread.__init__(self)
        self.tope = tope
```

Pero aún no es un hilo de ejecución separado, para esto hay que definir otra función, run :

```
    def run(self):
        print 'Ya soy un hilo
independiente'
        i = 0
        while i < self.tope:
            print i
            i = i+1
```

Para iniciar el hilo independiente hay que llamar a la función start del objeto (si, no run), y ya está.

```
>>> h = nuevo_hilo(20)
>>> h.start()
Ya soy un hilo independiente
>>>  # Aquí ya se separó el hilo
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

-1: Anexo

En estos momentos es posible que aún habiendo comprendido todo lo que se detalla aquí te preguntes como hacer algunas cosas, y es que esto no se escribió para ser una guía extensa sino como introducción a la programación, Python permite hacer casi cualquier cosa que se te ocurra:

Manipulación de paquetes a bajo nivel, gráficos 2D y 3D, cifrado y descifrado de datos, bluetooth ... entre las librerías estándar hay incluso unas para manejar fácilmente FTP, SMTP, archivos ZIP, archivos GZIP...

En [http://docs.python.org/release/2.6.6/library/index.html] se puede encontrar casi todo lo que se necesita saber sobre Python, pero a continuación dejo una lista de las cosas que creo que pueden resultar más interesantes:

http://docs.python.org/release/2.6.6/library/math.html: Librería matemática
http://docs.python.org/release/2.6.6/library/random.html: Generación de números pseudo-aleatorios
http://docs.python.org/release/2.6.6/library/os.html: Para hacer scripts más portables
http://docs.python.org/release/2.6.6/library/sys.html: Ejecutar otros programas y demás golosinas
http://docs.python.org/release/2.6.6/library/time.html: Esperas y medición de tiempo (¡ muy útil !)

http://docs.python.org/release/2.6.6/library/socket.html: Más conexiones de red
http://docs.python.org/release/2.6.6/library/ssl.html: Para conexiones cifradas :D
http://docs.python.org/release/2.6.6/library/simplehttpserver.html: Monta un servidor web en 6 líneas
http://docs.python.org/release/2.6.6/library/select.html: Para esperar a varios sockets

http://docs.python.org/release/2.6.6/library/sqlite3.html: Para usar bases de datos Sqlite3
http://docs.python.org/release/2.6.6/library/zipfile.html: Archivos ZIP
http://docs.python.org/release/2.6.6/library/gzip.html: Archivos GZIP
http://docs.python.org/release/2.6.6/library/hashlib.html: Funciones hash

http://docs.python.org/release/2.6.6/library/tkinter.html: Interfaces gráficas simples
http://docs.python.org/release/2.6.6/library/turtle.html: Gráficos de tortuga (para pasar el rato =P)